**DLL Injection**

DLL injection is a method that involves inserting a piece of code, structured as a Dynamic Link Library (DLL), into a running process. This technique allows the inserted code to run within the process's context, thereby influencing its behaviour or accessing its resources.

DLL injection finds legitimate applications in various areas. For instance, software developers leverage this technology for hot patching, a method that enables the amendment or updating of code seamlessly, without the need to restart the ongoing process immediately. A prime example of this is [Azure's use of hot patching for updating operational servers](https://learn.microsoft.com/en-us/azure/automanage/automanage-hotpatch#how-hotpatch-works), which facilitates the benefits of the update without necessitating server downtime.

Nevertheless, it's not entirely innocuous. Cybercriminals often manipulate DLL injection to insert malicious code into trusted processes. This technique is particularly effective in evading detection by security software.

There are several different methods for actually executing a DLL injection.

**LoadLibrary**

LoadLibrary is a widely utilized method for DLL injection, employing the LoadLibrary API to load the DLL into the target process's address space.

The LoadLibrary API is a function provided by the Windows operating system that loads a Dynamic Link Library (DLL) into the current process’s memory and returns a handle that can be used to get the addresses of functions within the DLL.

Code: c

#include <windows.h>

#include <stdio.h>

int main() {

// Using LoadLibrary to load a DLL into the current process

HMODULE hModule = LoadLibrary("example.dll");

if (hModule == NULL) {

printf("Failed to load example.dll\n");

return -1;

}

printf("Successfully loaded example.dll\n");

return 0;

}

The first example shows how LoadLibrary can be used to load a DLL into the current process legitimately.

Code: c

#include <windows.h>

#include <stdio.h>

int main() {

// Using LoadLibrary for DLL injection

// First, we need to get a handle to the target process

DWORD targetProcessId = 123456 // The ID of the target process

HANDLE hProcess = OpenProcess(PROCESS\_ALL\_ACCESS, FALSE, targetProcessId);

if (hProcess == NULL) {

printf("Failed to open target process\n");

return -1;

}

// Next, we need to allocate memory in the target process for the DLL path

LPVOID dllPathAddressInRemoteMemory = VirtualAllocEx(hProcess, NULL, strlen(dllPath), MEM\_RESERVE | MEM\_COMMIT, PAGE\_READWRITE);

if (dllPathAddressInRemoteMemory == NULL) {

printf("Failed to allocate memory in target process\n");

return -1;

}

// Write the DLL path to the allocated memory in the target process

BOOL succeededWriting = WriteProcessMemory(hProcess, dllPathAddressInRemoteMemory, dllPath, strlen(dllPath), NULL);

if (!succeededWriting) {

printf("Failed to write DLL path to target process\n");

return -1;

}

// Get the address of LoadLibrary in kernel32.dll

LPVOID loadLibraryAddress = (LPVOID)GetProcAddress(GetModuleHandle("kernel32.dll"), "LoadLibraryA");

if (loadLibraryAddress == NULL) {

printf("Failed to get address of LoadLibraryA\n");

return -1;

}

// Create a remote thread in the target process that starts at LoadLibrary and points to the DLL path

HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD\_START\_ROUTINE)loadLibraryAddress, dllPathAddressInRemoteMemory, 0, NULL);

if (hThread == NULL) {

printf("Failed to create remote thread in target process\n");

return -1;

}

printf("Successfully injected example.dll into target process\n");

return 0;

}

The second example illustrates the use of LoadLibrary for DLL injection. This process involves allocating memory within the target process for the DLL path and then initiating a remote thread that begins at LoadLibrary and directs towards the DLL path.

**Manual Mapping**

Manual Mapping is an incredibly complex and advanced method of DLL injection. It involves the manual loading of a DLL into a process's memory and resolves its imports and relocations. However, it avoids easy detection by not using the LoadLibrary function, whose usage is monitored by security and anti-cheat systems.

A simplified outline of the process can be represented as follows:

1. Load the DLL as raw data into the injecting process.
2. Map the DLL sections into the targeted process.
3. Inject shellcode into the target process and execute it. This shellcode relocates the DLL, rectifies the imports, executes the Thread Local Storage (TLS) callbacks, and finally calls the DLL main.

**Reflective DLL Injection**

Reflective DLL injection is a technique that utilizes reflective programming to load a library from memory into a host process. The library itself is responsible for its loading process by implementing a minimal Portable Executable (PE) file loader. This allows it to decide how it will load and interact with the host, minimising interaction with the host system and process.

[Stephen Fewer has a great GitHub](https://github.com/stephenfewer/ReflectiveDLLInjection) demonstrating the technique. Borrowing his explanation below:

"The procedure of remotely injecting a library into a process is two-fold. First, the library you aim to inject must be written into the target process’s address space (hereafter referred to as the 'host process'). Second, the library must be loaded into the host process to meet the library's runtime expectations, such as resolving its imports or relocating it to an appropriate location in memory.

Assuming we have code execution in the host process and the library we aim to inject has been written into an arbitrary memory location in the host process, Reflective DLL Injection functions as follows.

1. Execution control is transferred to the library's ReflectiveLoader function, an exported function found in the library's export table. This can happen either via CreateRemoteThread() or a minimal bootstrap shellcode.
2. As the library's image currently resides in an arbitrary memory location, the ReflectiveLoader initially calculates its own image's current memory location to parse its own headers for later use.
3. The ReflectiveLoader then parses the host process's kernel32.dll export table to calculate the addresses of three functions needed by the loader, namely LoadLibraryA, GetProcAddress, and VirtualAlloc.
4. The ReflectiveLoader now allocates a continuous memory region where it will proceed to load its own image. The location isn't crucial; the loader will correctly relocate the image later.
5. The library's headers and sections are loaded into their new memory locations.
6. The ReflectiveLoader then processes the newly loaded copy of its image's import table, loading any additional libraries and resolving their respective imported function addresses.
7. The ReflectiveLoader then processes the newly loaded copy of its image's relocation table.
8. The ReflectiveLoader then calls its newly loaded image's entry point function, DllMain, with DLL\_PROCESS\_ATTACH. The library has now been successfully loaded into memory.
9. Finally, the ReflectiveLoader returns execution to the initial bootstrap shellcode that called it, or if it were called via CreateRemoteThread, the thread would terminate."

**DLL Hijacking**

DLL Hijacking is an exploitation technique where an attacker capitalizes on the Windows DLL loading process. These DLLs can be loaded during runtime, creating a hijacking opportunity if an application doesn't specify the full path to a required DLL, hence rendering it susceptible to such attacks.

The default DLL search order used by the system depends on whether Safe DLL Search Mode is activated. When enabled (which is the default setting), Safe DLL Search Mode repositions the user's current directory further down in the search order. It’s easy to either enable or disable the setting by editing the registry.

1. Press Windows key + R to open the Run dialog box.
2. Type in Regedit and press Enter. This will open the Registry Editor.
3. Navigate to HKEY\_LOCAL\_MACHINE\\SYSTEM\\CurrentControlSet\\Control\\Session Manager.
4. In the right pane, look for the SafeDllSearchMode value. If it does not exist, right-click the blank space of the folder or right-click the Session Manager folder, select New and then DWORD (32-bit) Value. Name this new value as SafeDllSearchMode.
5. Double-click SafeDllSearchMode. In the Value data field, enter 1 to enable and 0 to disable Safe DLL Search Mode.
6. Click OK, close the Registry Editor and Reboot the system for the changes to take effect.

With this mode enabled, applications search for necessary DLL files in the following sequence:

1. The directory from which the application is loaded.
2. The system directory.
3. The 16-bit system directory.
4. The Windows directory.
5. The current directory.
6. The directories that are listed in the PATH environment variable.

However, if 'Safe DLL Search Mode' is deactivated, the search order changes to:

1. The directory from which the application is loaded.
2. The current directory.
3. The system directory.
4. The 16-bit system directory.
5. The Windows directory
6. The directories that are listed in the PATH environment variable

DLL Hijacking involves a few more steps. First, you need to pinpoint a DLL the target is attempting to locate. Specific tools can simplify this task:

1. Process Explorer: Part of Microsoft's Sysinternals suite, this tool offers detailed information on running processes, including their loaded DLLs. By selecting a process and inspecting its properties, you can view its DLLs.
2. PE Explorer: This Portable Executable (PE) Explorer can open and examine a PE file (such as a .exe or .dll). Among other features, it reveals the DLLs from which the file imports functionality.

After identifying a DLL, the next step is determining which functions you want to modify, which necessitates reverse engineering tools, such as disassemblers and debuggers. Once the functions and their signatures have been identified, it's time to construct the DLL.

Let’s take a practical example. Consider the C program below:

Code: c

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <windows.h>

typedef int (\*AddFunc)(int, int);

int readIntegerInput()

{

int value;

char input[100];

bool isValid = false;

while (!isValid)

{

fgets(input, sizeof(input), stdin);

if (sscanf(input, "%d", &value) == 1)

{

isValid = true;

}

else

{

printf("Invalid input. Please enter an integer: ");

}

}

return value;

}

int main()

{

HMODULE hLibrary = LoadLibrary("library.dll");

if (hLibrary == NULL)

{

printf("Failed to load library.dll\n");

return 1;

}

AddFunc add = (AddFunc)GetProcAddress(hLibrary, "Add");

if (add == NULL)

{

printf("Failed to locate the 'Add' function\n");

FreeLibrary(hLibrary);

return 1;

}

HMODULE hLibrary = LoadLibrary("x.dll");

printf("Enter the first number: ");

int a = readIntegerInput();

printf("Enter the second number: ");

int b = readIntegerInput();

int result = add(a, b);

printf("The sum of %d and %d is %d\n", a, b, result);

FreeLibrary(hLibrary);

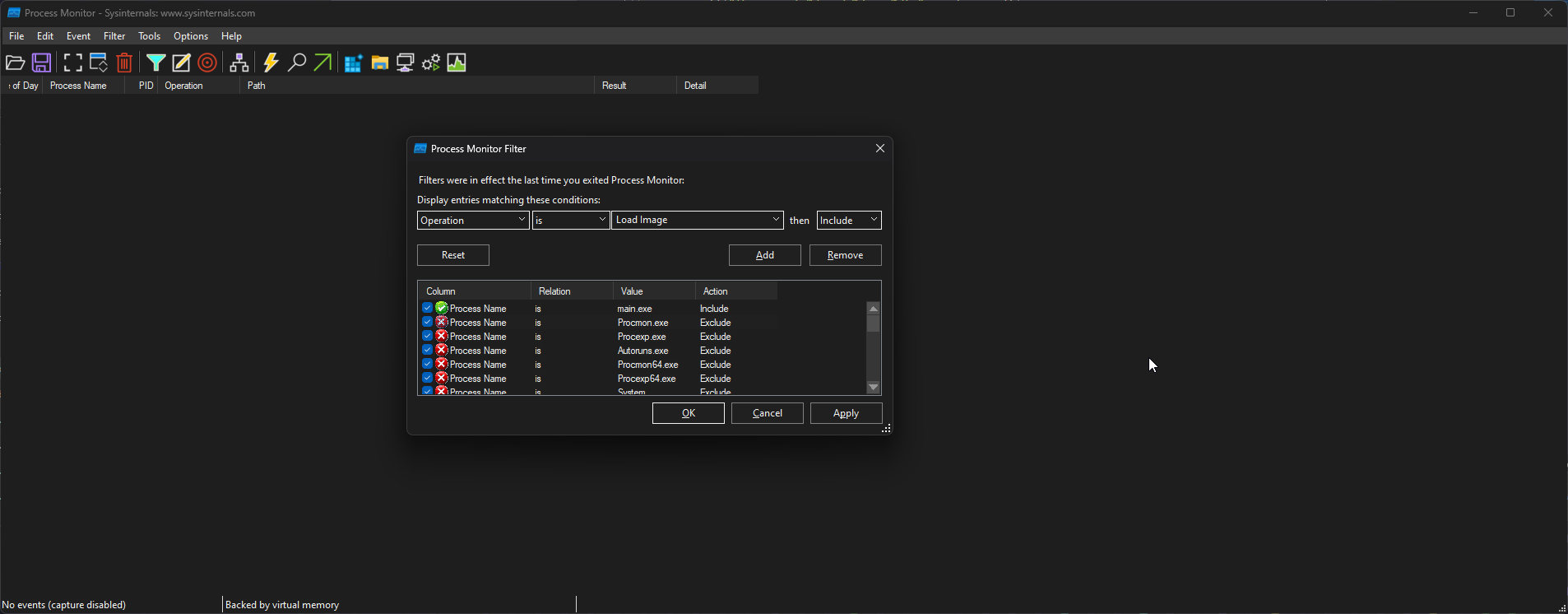
system("pause");

return 0;

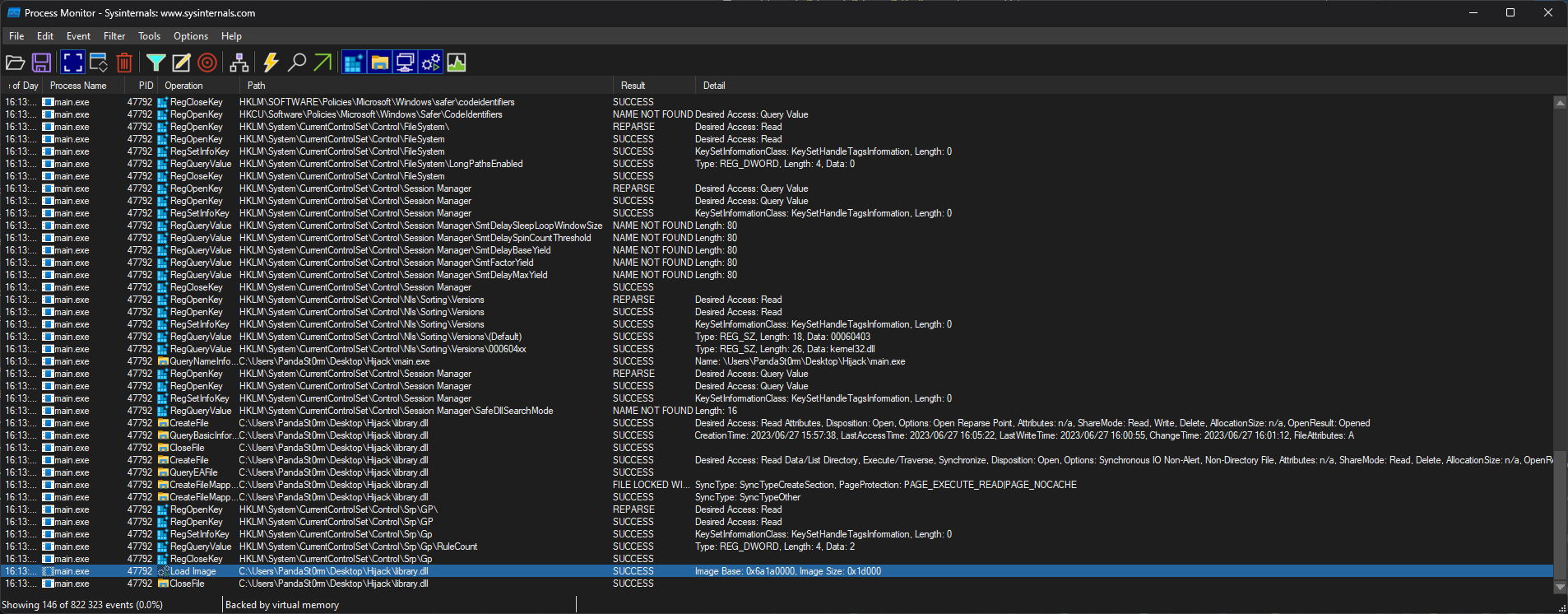
}

It loads an add function from the library.dll and utilises this function to add two numbers. Subsequently, it prints the result of the addition. By examining the program in Process Monitor (procmon), we can observe the process of loading the library.dll located in the same directory.

First, let's set up a filter in procmon to solely include main.exe, which is the process name of the program. This filter will help us focus specifically on the activities related to the execution of main.exe. It is important to note that procmon only captures information while it is actively running. Therefore, if your log appears empty, you should close main.exe and reopen it while procmon is running. This will ensure that the necessary information is captured and available for analysis.



Then if you scroll to the bottom, you can see the call to load library.dll.



We can further filter for an Operation of Load Image to only get the libraries the app is loading.

16:13:30,0074709 main.exe 47792 Load Image C:\Users\PandaSt0rm\Desktop\Hijack\main.exe SUCCESS Image Base: 0xf60000, Image Size: 0x26000

16:13:30,0075369 main.exe 47792 Load Image C:\Windows\System32\ntdll.dll SUCCESS Image Base: 0x7ffacdbf0000, Image Size: 0x214000

16:13:30,0075986 main.exe 47792 Load Image C:\Windows\SysWOW64\ntdll.dll SUCCESS Image Base: 0x77a30000, Image Size: 0x1af000

16:13:30,0120867 main.exe 47792 Load Image C:\Windows\System32\wow64.dll SUCCESS Image Base: 0x7ffacd5a0000, Image Size: 0x57000

16:13:30,0122132 main.exe 47792 Load Image C:\Windows\System32\wow64base.dll SUCCESS Image Base: 0x7ffacd370000, Image Size: 0x9000

16:13:30,0123231 main.exe 47792 Load Image C:\Windows\System32\wow64win.dll SUCCESS Image Base: 0x7ffacc750000, Image Size: 0x8b000

16:13:30,0124204 main.exe 47792 Load Image C:\Windows\System32\wow64con.dll SUCCESS Image Base: 0x7ffacc850000, Image Size: 0x16000

16:13:30,0133468 main.exe 47792 Load Image C:\Windows\System32\wow64cpu.dll SUCCESS Image Base: 0x77a20000, Image Size: 0xa000

16:13:30,0144586 main.exe 47792 Load Image C:\Windows\SysWOW64\kernel32.dll SUCCESS Image Base: 0x76460000, Image Size: 0xf0000

16:13:30,0146299 main.exe 47792 Load Image C:\Windows\SysWOW64\KernelBase.dll SUCCESS Image Base: 0x75dd0000, Image Size: 0x272000

16:13:31,7974779 main.exe 47792 Load Image C:\Users\PandaSt0rm\Desktop\Hijack\library.dll SUCCESS Image Base: 0x6a1a0000, Image Size: 0x1d000

**Proxying**

We can utilize a method known as DLL Proxying to execute a Hijack. We will create a new library that will load the function Add from library.dll, tamper with it, and then return it to main.exe.

1. Create a new library: We will create a new library serving as the proxy for library.dll. This library will contain the necessary code to load the Add function from library.dll and perform the required tampering.
2. Load the Add function: Within the new library, we will load the Add function from the original library.dll. This will allow us to access the original function.
3. Tamper with the function: Once the Add function is loaded, we can then apply the desired tampering or modifications to its result. In this case, we are simply going to modify the result of the addition, to add + 1 to the result.
4. Return the modified function: After completing the tampering process, we will return the modified Add function from the new library back to main.exe. This will ensure that when main.exe calls the Add function, it will execute the modified version with the intended changes.

The code is as follows:

Code: c

// tamper.c

#include <stdio.h>

#include <Windows.h>

#ifdef \_WIN32

#define DLL\_EXPORT \_\_declspec(dllexport)

#else

#define DLL\_EXPORT

#endif

typedef int (\*AddFunc)(int, int);

DLL\_EXPORT int Add(int a, int b)

{

// Load the original library containing the Add function

HMODULE originalLibrary = LoadLibraryA("library.o.dll");

if (originalLibrary != NULL)

{

// Get the address of the original Add function from the library

AddFunc originalAdd = (AddFunc)GetProcAddress(originalLibrary, "Add");

if (originalAdd != NULL)

{

printf("============ HIJACKED ============\n");

// Call the original Add function with the provided arguments

int result = originalAdd(a, b);

// Tamper with the result by adding +1

printf("= Adding 1 to the sum to be evil\n");

result += 1;

printf("============ RETURN ============\n");

// Return the tampered result

return result;

}

}

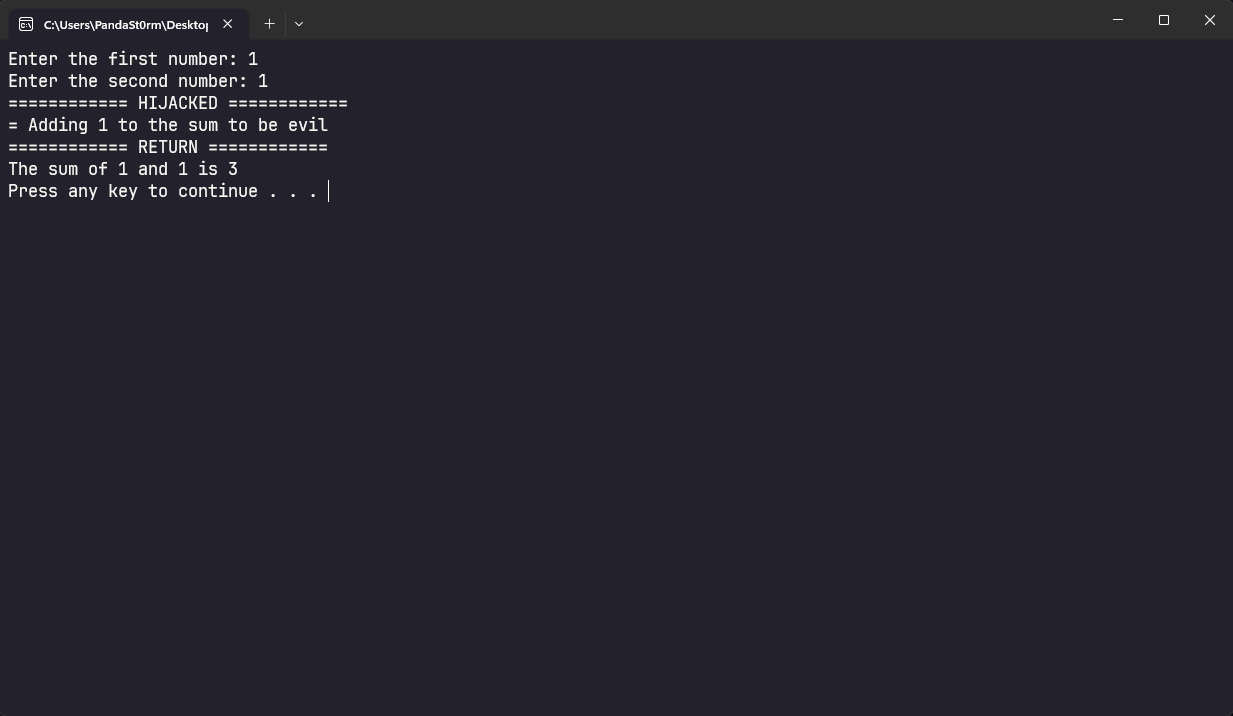
// Return -1 if the original library or function cannot be loaded

return -1;

}

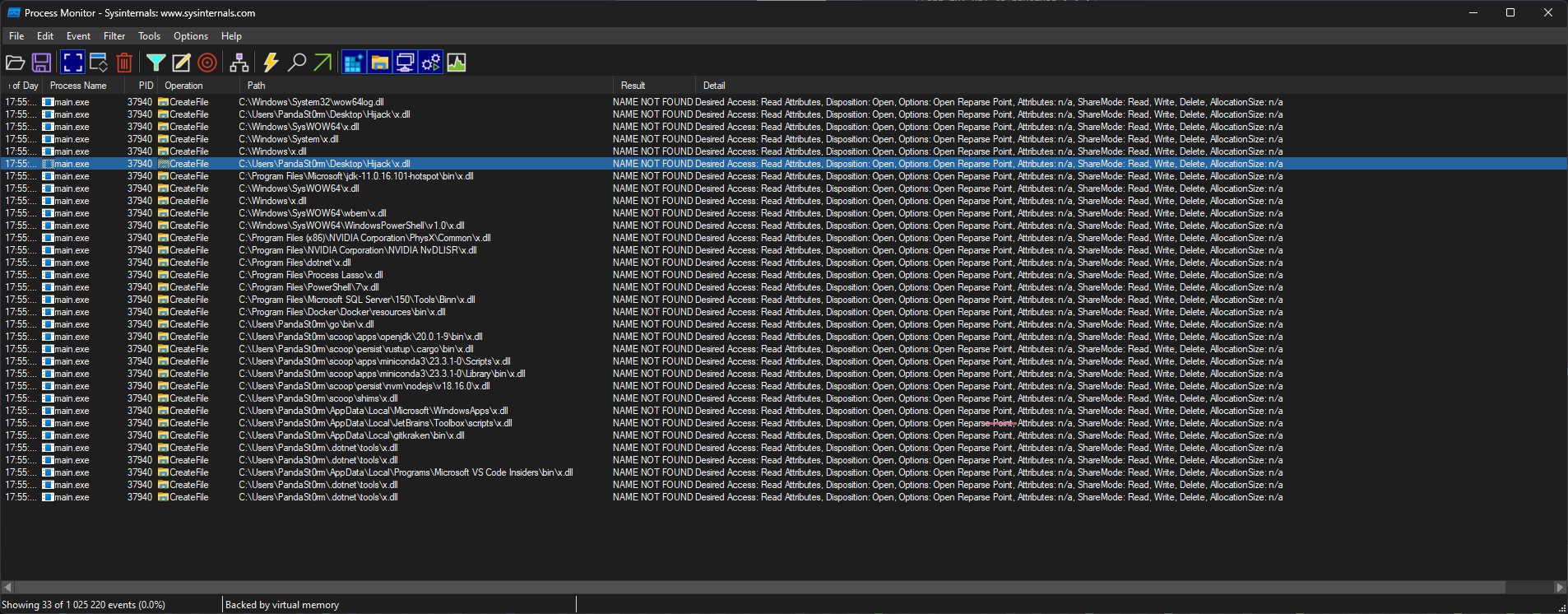
Either compile it or use the precompiled version provided. Rename library.dll to library.o.dll, and rename tamper.dll to library.dll.

Running main.exe then shows the successful hack.



**Invalid Libraries**

Another option to execute a DLL Hijack attack is to replace a valid library the program is attempting to load but cannot find with a crafted library. If we change the procmon filter to focus on entries whose path ends in .dll and has a status of NAME NOT FOUND we can find such libraries in main.exe.



As we know, main.exe searches in many locations looking for x.dll, but it doesn’t find it anywhere. The entry we are particularly interested in is:

17:55:39,7848570 main.exe 37940 CreateFile C:\Users\PandaSt0rm\Desktop\Hijack\x.dll NAME NOT FOUND Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Attributes: n/a, ShareMode: Read, Write, Delete, AllocationSize: n/a

Where it is looking to load x.dll from the app directory. We can take advantage of this and load our own code, with very little context of what it is looking for in x.dll.

Code: c

#include <stdio.h>

#include <Windows.h>

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul\_reason\_for\_call, LPVOID lpReserved)

{

switch (ul\_reason\_for\_call)

{

case DLL\_PROCESS\_ATTACH:

{

printf("Hijacked... Oops...\n");

}

break;

case DLL\_PROCESS\_DETACH:

break;

case DLL\_THREAD\_ATTACH:

break;

case DLL\_THREAD\_DETACH:

break;

}

return TRUE;

}

This code defines a DLL entry point function called DllMain that is automatically called by Windows when the DLL is loaded into a process. When the library is loaded, it will simply print Hijacked... Oops... to the terminal, but you could theoretically do anything here.

Either compile it or use the precompiled version provided. Rename hijack.dll to x.dll, and run main.exe.

